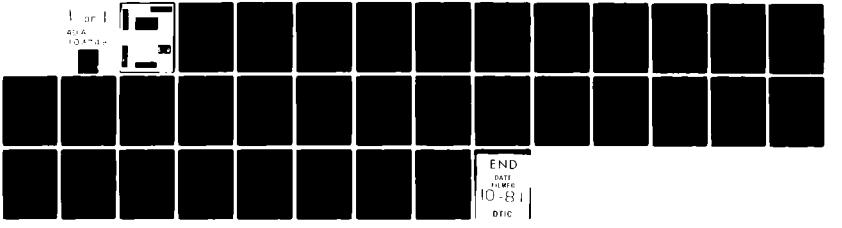
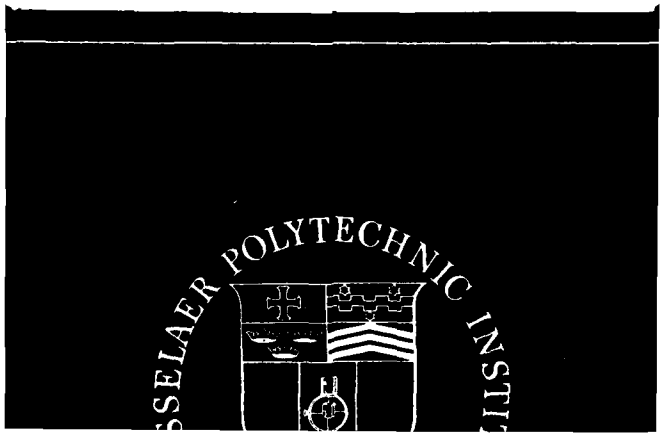
AD-A103 749 RENSSELAER POLYTECHNIC INST TROY NY DEPT OF MATHEMAT--ETC F/G 9/2
THEOREM GENERALIZATION IN PROGRAM VERIFICATION.(U)
JUN 81 J VYTOPIL, S K ABDALI N00014-75-C-1026
UNCLASSIFIED RPI-CS-8102 NL



END
DATE FILMED
10-81
DTIC


POLYTECHNIC

(14) RPI-

Technical Report CS-8102

(6) THEOREM GENERALIZATION IN PROGRAM VERIFICATION.

(10) Jan Mytopil
S. Kamal Abdali

(11) June 1981

(12) 33

(9) Technical Rept.

Prepared for

U.S. Office of Naval Research
Contract Number (15) N00014-75-C-1026

Mathematical Sciences Department
Rensselaer Polytechnic Institute
Troy, New York 12181

1. INTRODUCTION

This paper deals with the generalization of theorems arising in program verification. The theorems of concern here are mostly about the properties of recursively defined functions (derived from program definitions), and their proof involves the use of induction. In proving such theorems, we have to confront three questions: How to find the variables or structures on which to carry out induction, how to choose an appropriate induction schema, and how to find its correct instantiation.

Intuitively, recursion and induction are complementary (see Boyer and Moore [2]): Recursion starts with some structure and decomposes it until some basic structure is obtained; induction starts with some basic structure and builds up. This duality can be used for solving the problems of setting up the induction: we have to induct on those structures that are being recursively decomposed by the function ([1,2]).

In trying to prove theorems about recursively defined functions, it often happens that the induction step fails, that is, the theorem is not strong enough to carry itself through the induction. In such cases we try to find a more general theorem, which should be easier to prove than the one in which we are actually interested. The discovery of helpful generalizations requires a deep understanding of the function's performance. Some heuristics for generalization are based on the analysis of the functions involved: we try to understand the roles being played by the arguments in the computation of the function, and we try to use this knowledge for generalization. This type of heuristics is exemplified by the method of Greif and Waldinger [4], where the symbolic execution of the program for its first few terms is followed by pattern matching to find a closed form expression which generates the series so obtained. Other types of heuristics are represented by the method discovered by Boyer and Moore [2]. Here, expressions common to both sides of equality are replaced with a new variable. It is quite a simple and powerful heuristics that seems to work well for simple list-manipulating functions.

Consider the programs containing iterations. In such programs, some variables are used to accumulate results and are usually initialized with constant values. In the functional definitions obtained from such programs, these variables turn into auxiliary arguments with constant values, or, in the words of Wegbreit [11], specialized arguments. These arguments do not play any role in the functional definition and yet their contribution to the function computation is essential. The effect of their replacement by variable arguments must be understood well if we are to carry out by induction the proof of

some property relevant to such a function.

Since we are given some functional definition and some theorems about the functions so defined, we can, possibly, change either of these to make induction work. Hence we will study two ways to handle the problem caused by argument specialization:

Theorem Generalization Replacing the constant argument by a variable such that a more general theorem than the given one may be provable by induction. We describe how to generalize theorems about two typical classes of functions.

Function Redefinition While the above method removes a specialization by creating a new, stronger theorem, function redefinition uses the specialization by creating a new definition from which all specialized arguments are effectively deleted. This requires a total rearrangement of the computation: it is not a syntactic manipulation. We describe the redefinition procedure for two classes of programs.

There is a duality between the problem of finding invariant assertions and the problem of finding theorem generalization. It is shown in [10,11] that for certain class of functions and theorems, these problems are indeed equivalent: one can get an invariant assertion if one knows the generalization, and vice versa. As a side-benefit of our generalization strategy, we show how to obtain invariant assertions in certain cases.

2. BASIC NOTATION AND DEFINITIONS

For defining functions we use the notation of recursive schemas (Manna [5]).

Our usage of letters is as follows:

F,G,H,f,g,h	to denote functions
x,y,z,u,v,w	to denote variables
p,q	to denote variables
c	to denote constants

Subscripts and superscripts may be also added to the letters.

Furthermore we use

IF p THEN e1 ELSE e2	for conditional expressions
e1 <= e2	for defining functions (or predicates) e1 by the expression e2

Subscripts and superscripts may be also used.

Basically, functions are either instances of the recursion schema

```
F(y) <= IF p(y) THEN f(y)
                ELSE h(y,F(g(y)))
```

or of the composition schema

```
F(y) <= h(y,g(y))
```

where f,g,h are previously defined functions.

Depending on the domain there is a number of basic functions and basic constants; for example:

List theory

basic functions: CONS,CDR,CAR,EQUAL,ATOM
basic constant: NIL

Number theory

basic functions: SUCC,PRED,=
basic constant: 0

Sometimes we will use well-known arithmetic functions and abbreviations without giving their explicit definitions.

In general we use unsubscripted letters to denote vectors and subscripted ones to denote the components of a vector. We use angular brackets to denote the explicit formation of a vector, and indexing for decomposition. Thus, x denotes the vector of variables <x1, x2,...,xn> for some fixed n. Further, we reserve letters x,y,z for the following special purpose: function F takes the input vector x, computes on the auxiliary state vector y and returns the output vector z = F(x) as the function value. c denotes a constant vector.

We will not define how to obtain the value of a function when applied to an argument. For our purpose, an intuitive understanding of this process is sufficient (for a detailed discussion see Manna and Pnuelli [6]).

An important subclass of functions is obtained by instantiations of the iterative schema

```
F(y,y') <= IF p(y) THEN y'                    (1)
                   ELSE F(f(y),g(y,y'))
```

This type of functions is obtained when we translate simple programs from a language with iteration (see McCarthy[7]). For instance, the program schema:

```
WHILE p(y) DO (y,y') :=: (f(y),g(y,y')) OD;        (2)
RESULT :=: y'
```

where :=: denotes concurrent assignment, will be translated into the above-mentioned function.

The variable y tested in the termination condition is a recursion variable. Following Moore [9], y' is called an accumulator because "if a function modifies an argument during (some) recursive call but does not test the argument in the termination condition, the program considers that variable to be an accumulator".

3. THEOREM GENERALIZATION

3.1 Generality of Theorems

Suppose that we wish to prove by induction a theorem Th about the function F, defined over the domain of natural numbers, as an instance of the recursion schema

```
F(y) <= IF p(y) THEN g(y) ELSE h(y,F(n(y)))        (3)
```

To prove Th(F(x)) by mathematical induction we have to prove

1) Basis: Th(F(0))
2) Induction step: Th(F(x)) -> Th(F(Succ(x))

Usually the basis can be proven by evaluating F(0), but the induction step turns out to be more difficult. To simplify the conclusion in the induction step, we may evaluate F(Succ(x)):

```
F(Succ(x)) = IF p(Succ(x)) THEN g(Succ(x))
                           ELSE h(Succ(x),F(n(Succ(x))))
```

Because the hypothesis is about F(x), namely Th(F(x)), we have to express F(n(Succ(x))) in terms of F(x), and if n is different from the predecessor function it is not at all obvious what to do next. So we can see that mathematical induction, which serves best for proofs of theorems about functions defined by primitive recursive schema, is not always suitable for proofs of theorems about functions defined by recursive schema (3). For carrying out inductive proofs for functions defined by the recursive schema (3), we must, in order to simplify the conclusion in the

induction step, take into the account the way in which the function F is computed.

Let us observe the evaluation of F(x):

$$F(x) = h(x,F(n(x))) = h(x,h(n(x),F(n(n(x)))))$$

$$=...=h(x,h(n(x),...,h(n^{(k-1)},F(n^{k}(x)))))$$

The values of F form the sequence

$$x,\ n(x),\ n(n(x)),...,n^{k}(x)$$

where $k=\min\{j \mid p(n^{j}(x))\}$.

Thus we can use the following induction rule to prove Th(F(x)).

Basis: $Th(F(n^{k}(x)))$

Induction step: $Th(F(n^{(j+1)}(x))) \rightarrow Th(F(n^{j}(x)))$ for all $j<k$

The schema (3) and the relation $k=\min\{j \mid p(n^{j}(x))\}$ allow us to obtain the following simpler and stronger rule, that we will refer to as the case induction rule.

Case Induction Rule

Let a total function F be defined as an instance of the recursion schema

$$F(y) \Leftarrow \text{IF } p(y) \text{ THEN } g(y) \text{ ELSE } h(y,F(n(y)))$$

Then a theorem Th(F(x)) about F holds if and only if both of the following conditions are satisfied:

a) Basis: $p(y) \rightarrow Th(F(y))$ (4)
b) Induction step: $\neg p(y) \ \&\ Th(F(n(y))) \rightarrow Th(F(y))$

If function F is a total function then the conditions a) and b) are not only sufficient, as follows from the above discussion, but also necessary. This can be proved by contradiction as follows:

Suppose Th(F(x)) holds for all x, but either a) or b) do

not hold, then

If p(y) -> Th(F(y)) is false for some y0 then Th(F(y0)) must be false, since Th(F(y0)) is defined for all y. This contradicts the assumption that Th(F(x)) is true for all x.

If ¬p(y) & Th(F(n(y))) -> Th(F(y)) is false for a y0, Th(F(y0)) must be false, resulting in the same contradiction.

Thus, if the function F is total, the conditions a) and b) are not only sufficient but also necessary in order for Th(F(x)) to be true.

For functions defined by the recursion schema, the case induction rule (4) is easier to use than mathematical induction, because the evaluation of F(y) on the right-hand side of the implication gives us the expression

¬p(y)&Th(F(n(y))) -> Th(IF p(y) THEN g(y) ELSE h(y,F(n(y))))

Since both sides of implication now contain the expression F(n(y)), it seems to be easier to simplify the conclusion using the hypothesis. This is why in the rest of this chapter, we use the case induction rule (4) to prove theorems by induction.

Suppose we have to prove the theorem

IN(x) -> F(x) = G(x)

where F is a function defined by an instance of the recursion schema (3), G is some other previously defined function and IN is a predicate describing the initial values of arguments of function F. Using case induction we have to prove

Basis:

p(x) -> (IN(x) -> F(x) = G(x))

which can be simplified into

p(x) & IN(x) -> g(x) = G(x).

Induction step:

[¬p(x)&(IN(n(x)) -> F(n(x))=G(n(x)))] ->
[(IN(x) -> F(x) = G(x))]

which can be simplified into

[¬p(x)&IN(x)&(IN(n(x))->F(n(x))=G(n(x)))]->[F(x)=G(x)]

or

$$[p(x) \;\&\; IN(x) \;\&\; (IN(n(x)) \rightarrow F(n(x)) = G(n(x)))] \rightarrow [h(x,F(n(x))) = G(x)]$$

To simplify the conclusion $h(x,F(n(x))) = G(x)$, we have to use the assumption $IN(n(x)) \rightarrow F(n(x)) = G(n(x))$. Now, if $IN(n(x))$ is not true, then we do not know anything about the relation between $F(n(x))$ and $G(n(x))$ to simplify the conclusion. But if $IN(n(x))$ happens to be true because of $\neg p(x)$ and $IN(x)$, then the assumption $F(n(x)) = G(n(x))$ can be used in simplifying the conclusion, and the theorem to prove would be

$$\neg p(x) \;\&\; IN(x) \rightarrow h(x,G(n(x))) \rightarrow G(n(x)).$$

Thus if $\neg p(x) \;\&\; IN(x) \rightarrow IN(n(x))$ holds, then we can use the assumption to simplify the conclusion. In other words the theorem in this case is strong enough to carry itself through the induction.

To sum up the discussion, we state the following theorem:

THEOREM 1: If a predicate IN satisfies the condition

$$IN(x) \;\&\; \neg p(x) \rightarrow IN(n(x)) \qquad (5)$$

and the function F defined by (3) above is total on the domain specified by IN, then the property

$$IN(x) \rightarrow F(x) = G(x)$$

holds iff both

a) $p(x) \;\&\; IN(x) \rightarrow g(x) = G(x)$
b) $\neg p(x) \;\&\; IN(x) \rightarrow h(x,G(n(x))) = G(x)$

are true.

PROOF:

<-) By case induction.

Basis: We have to prove that

$$p(x) \;\&\; IN(x) \rightarrow F(x)=G(x)$$

but this is immediate from the definition of F.

Induction step: We have to prove that

```
      [¬p(x)&(IN(n(x))->G(n(x))=F(n(x)))]->[IN(x)->G(x)=F(x)]

 1  ¬p(x) & IN(x) -> h(x,G(n(x))) = G(x)    assumption B
 2  ¬p(x)&IN(n(x))&G(n(x)) = F(n(x)) -> h(x,F(n(x))) = G(x)
                  from 1 using properties of -> and &
 3  ¬p(x) & [IN(x) & IN(n(x)) & G(n(x)) = F(n(x))]  ->
                                          [h(x,F(n(x))) = G(x)]
                  from 2 and (5) using properties of -> and &
 4  ¬p(x)&[IN(n(x))->G(n(x))=F(n(x))] -> [IN(x)->G(x)=F(x)]
       from 3 and def. of F, using properties of -> and &
```

->) By case analysis

CASE 1: p(x) is true.

We have to prove that

$$p(x) \;\&\; IN(x) \rightarrow g(x) = G(x)$$

But this is immediate from the definition of F.

CASE 2: ¬p(x) is true.

We have to prove

$$\neg p(x) \;\&\; IN(x) \rightarrow h(x,G(n(x))) = G(x)$$

```
 1  ¬p(x) & IN(x) -> h(x,F(n(x)) = G(x)
                    from assumptions and def. of F using
                         properties of & and ->
 2  ¬p(x) & IN(n(x)) -> ¬p(x) & F(n(x)) = G(n(x))
    from asumptions with instantiation of x as n(x) using
        properties of & and  ->
 3  ¬p(x) & IN(x) -> ¬p(x) & IN(n(x))
                   from (5) using properties of & and  ->
 4  ¬p(x) & IN(x) ->¬p(x) & F(n(x)) = G(n(x))
                                        from 3 and and 2
 5  ¬p(x) & IN(x) -> ¬p(x) & G(n(x)) = F(n(x))
                                    & h(x,F(n(x))) = G(x)
                from 4 and 1 using properties of -> and &
 6  ¬p(x) & IN(x) -> h(x,G(n(x))) = G(x)
                    from 5 using properties of ->and &.
```

Essentially, the condition

$$\neg p(x) \;\&\; IN(x) \rightarrow IN(n(x))$$

is a requirement that IN, the input specification, is strong enough to describe all possible values that the arguments of F can take on during the computation. For example, consider the

case of the theorem

$$F(x,1) = x!$$

where we define

$$F(y1,y2) \Leftarrow IF\ y1 = 0\ THEN\ 1\ ELSE\ F(y1-1,\ y1*y2)$$

In this case

$$p(y1,y2) \Leftarrow y1=0$$
$$IN(y1,y2) \Leftarrow y1\ is\ INT\ \&\ y2 = 1$$
$$n(y1,y2) \Leftarrow \langle y1-1, y1*y2\rangle$$
$$IN(y1,y2) = (y1\ is\ INT\ \&\ y2 = 1)$$

where INT denotes the domain of integers. Condition (5) is not satisfied because

$$\neg p(x)\ \&\ IN(x) \rightarrow IN(n(x))$$

or

$$[y1\ is\ INT\ \&\ y2=1\ \&\ (y1\neq 0)] \rightarrow [(y1-1)\ is\ INT\ \&\ y1*y2=1]$$

or

$$[y1\ is\ INT\ \&\ y2=1\ \&\ (y1\neq 0)] \rightarrow [(y1-1)\ is\ INT\ \&\ y1*1=1]$$

is not true. The reason is that y2 is too specialized.

But suppose the theorem and the input specification were to be generalized to be, respectively

$$F(x,y) = y*x!$$
$$IN(y1,y2) \Leftarrow [y1\ is\ INT\ \&\ y2\ is\ INT]$$

Now condition (5) below clearly holds:

$$IN(y1,y2) = (y1\ is\ INT\ \&\ y2\ is\ INT)$$
$$[y1\neq 0\ \&\ (y1\ is\ INT\ \&\ y2\ is\ INT)] \rightarrow [(y1-1)\ is\ INT\ \&\ (y1*y2)\ is\ INT]$$

Thus, the input specification IN, must be general enough to satisfy (5). In other words (5) is the criterion for judging whether a certain theorem generalization strategy is useful. If, using the specific generalization strategy, we make it more likely to satisfy condition (5), then that strategy is useful. We do not intend to check whether a specific theorem is strong enough; this we leave to the proof process itself. Our theorem is a special case of the theorem given by Wegbreit [11] but is a more general version than the one given by Misra [8]. In their

papers it was shown moreover that if a function F, total on the domain specified by IN, satisfies the conditions

1) IN(x) -> F(x) = G(x)
2) ¬p(x) & IN(x) -> IN(n(x))

then a correct invariant assertion is

IN(y) & G(x) = G(y)

Thus, to obtain the invariant assertion for a program in which the input specification is not strong enough, we can first use a generalization strategy to strengthen IN and then construct the invariant assertion.

3.2 Generalization Scheme I

Suppose we have to prove

$$F(x1,1) = x1! \qquad (6)$$

where F is defined:

F(y1,y2) <= IF y1=0 THEN y2 ELSE F(y1-1,y1*y2)

To prove this theorem by induction we have to establish

Basis: F(0,1) = 0!
Induction Step: F(x1,1) = x1! -> F(x1+1,1) = (x1+1)!

For basis, we just start with the left-hand-side and substitute the definition of F.

F(0,1) = IF (0=0) THEN 1 ELSE F(0-1,0*1) = 1=0!

This verifies the basis. Now let us try the induction step.

F(x1+1,1) = IF (x1+1=0) THEN 1 ELSE F(x1+1-1,(x1+1)*1)
= F(x1,x1+1),

since x1+1≠0 for any natural number. But now we realize that we are stuck as the induction hypothesis does not help us. It would let us simplify F(x1,1) but not F(x1,x1+1).

But instead of proving (6) directly, let us try to prove its following generalization.

$$F(x1,x2) = x1*x2$$

The proof by induction on x1 goes very smoothly. For the basis,

we just use the definition of F to obtain

$$F(0,x2) = IF\ (0=0)\ THEN\ x2\ ELSE\ F(x1-1,0*x2) = x2$$
$$= 1*x2 = 0!*x2$$

For the induction step, we assume F(x1,x2) = x1*x2, and then we try to prove F(x1+1,x2) = (x1+1)! * x2.

$$F(x1+1,x2) = IF\ (x1+1=0)\ THEN\ x2\ ELSE\ F(x1+1-1,(x1+1)*x2)$$
$$= F(x1,(x1+1)*x2),\ since\ x1+1\neq 0\ for\ any\ natural\ number$$
$$= x1!*(x1+1)*x2,\ by\ induction\ hypothesis$$
$$= (x1+1)!*x2$$

Hence the theorem F(x1,x2) = x1!*x2 has been established. The weaker theorem F(x1,1) = x1! is just the special case x2=1 of the theorem F(x1,x2) = x1!*x2.

The situation described above is quite common. In a program using iteration, some of the arguments are used as help arguments. Their initial values are usually constant and therefore they will play nc role in the original description of the function properties, yet their contribution to the computation is essential and must be understood and expressed in the theorems about function properties. Ideally, the initial values of arguments of a function should be as mutually independent and as general as possible.

In general, suppose we have to prove

$$F1(x,c) = F2(x) \qquad (7)$$

where F1 is an instance of iteration schema

$$F(y,y') <= IF\ p(y)\ THEN\ y'\ ELSE\ F(f(y),g(y,y'))$$

Then we observe that the final value of y' (and therefore the value of F1(x,c)), is built up by repeated application of the function g. The final value of y' is thus

$$t(i) = g(f^{i}(x),\ g(f^{(i-1)}(x),.....g(f(x),g(x,c)))$$

where f^{i} denotes f(f(f...(x))) with i applications of f. Suppose we like to generalize (7) by replacing c with h1(z,c) where z is a new variable. Then the final value of y' will be

$$g(f^{i}(x), g^{(i-1)}(f(x),\ldots,g(x,h1(c,z))))$$

Now suppose we can find two functions h1 and h2 with the property

$$g(u,h1(v,w)) = h1(g(u,v),h2(u,v,w)) \qquad (8)$$

Then

$$g(u,g(u',h1(v,w))) = g(u,h1(g(u',v),h2(u',v,w))) = h1(g(u,g(u',v)),h2(u,v,h2(u,v,w)))$$

In other words if we can find a pair of functions h1 and h2 with the property (8) then the final value of y' is

$$h1(t(i), h2(f^{i}(x),t(i-1),h2(f^{(i-1)}(x),t(i-1),\ldots, h2(f^{2}(x),t(1),h2(f(x),g(x,c),h2(x,c,z))))) \qquad (9)$$

The first argument of h1, t(i) is equal to F1(x,c), and the second argument of h1 is the iteration of h2 with the first and second arguments having the same values as they have during the evaluation of F1. Using the definition of F1, we can define a new function F3 so that F3(x,c,z) equals the second argument of h1 in (9). This F3 is defined as follows.

```
F3(y,y',y'') <= IF p1(y) THEN y''
                  ELSE F3(f(y),g(y,y'),h2(y,y',y''))
```

Now the generalization of a theorem should be an expression which

a) we believe is in fact a theorem
b) has the original theorem as an instance
c) is easier to prove

We suggest

$$F1(x,h1(c,z)) = h1(F2(x),F3(x,c,z))$$

as a generalization of (7). It has all the above mentioned properties, as the following theorem shows.

THEOREM 2: Let g,f,F2 be some previously defined functions, F1 be defined by the iteration schema

```
F1(y,y') <= IF p(y) THEN y' ELSE F1(f(y),g(y,y'))
```

and F3 be defined by

```
F3(y,y'y'')<= IF p(y) THEN y''
                      ELSE F3(f(y),g(y,y'),h2(y,y'y''))
```

If

```
F1(x,c) = F2(x)
```

and there exist functions h1 and h2 satisfying

```
g(u,h1(v,w)) = h1(g(u,v),h2(u,v,w))
```

then

```
F(x,h1(c,z)) = h1(F2(x),F3(x,c,z))                    (10)
```

holds.

LEMMA 1: Under the conditions and definitions of THEOREM 2, it is the case that

```
F1(x,h1(w,z)) = h1(F1(x,w),F3(x,w,z))
```

PROOF: By case induction.

Basis: We have to prove

```
p(x) -> h1(w,z)=h1(w,F3(x,w,z))
```

This we do as follows

```
1  p(x)                          assumption
2  F1(x,h1(w,z)) = h1(w,z)       from 1 and def. of F1
3  = h1(F1(x,w),F3(x,w,z))
                          from 1 and def. of F1 and F3
```

Induction step: We have to prove

```
[¬p(x) & F1(f(x),h1(w,z)) = h1(F1(f(x),F3(f(x),w,z))]
    -> [F(x,h1(w,z))= h1(F1(x,F3(x,w,z)))]
```

This we do as follows:

```
1  ¬p(x) & F1(f(x),h1(w,z)) = h1(F1(x,w),F3(x,w,z))
                              assumption
2  F1(x,h1(w,z)) = F1(f(x),g(x,h1(w,z)))
                          from 1 and def. of F1
3  = F1(f(x),h1(g(x,w),h2(x,w,z)))
```

from (8)

4 = h1(F1(f(x),g(x,w)),F3(f(x),g(x,w),h2(x,wz)))
from 1 (instantiate w as g(x,w) and z as (f2(x,w,z))

5 = h1(F1(x,w),F3(x,w,z))
from def. of F1 and F3

PROOF OF THEOREM 2

1 F1(x,h1(c,z)) = h1(F1(x,c),F3(x,c,z))
from LEMMA 1 (instantiate w as c)

2 F1(x,h1(c,z)) = h1(F2(x),F3(x,c,z))
from 1 and assumption F1(x,c)=F2(x)

The condition

$$g(u,h1(v,w)) = h1(g(u,v),h2(u,v,w))$$

does not guarantee that the original expression is an instance of the original theorem. In general, it is difficult to state how to derive h1 and h2 so that the original theorem is an instance of the more general expression (10). Nevertheless, we can say what a sufficcient condition is.

THEOREM 3: Under the definitions and conditions of the THEOREM 2, a sufficient condition that F1(x,c) = F2(x) is an instance of F1(x,h1(c,z) = h1(F2(x),F3(x,c,z)) is that for all u and v, there exists a z such that

$$(h1(u,z) = u \;\&\; h2(u,v,z) = z) \qquad (11)$$

PROOF :

Let z0 be the value of z satisfying (11). Then

$$F1(x,h1(c,z0)) = F1(x,c)$$
$$h1(F2(x),F3(x,c,z0)) = h1(F2(x),z0) = F2(x)$$

Thus, for z=z0, F1(x,h1(c,z))=h1(F2(x),F3(x,c,z)) can be simplified to F1(x,c)=F2(x).

Although (11) is not a necessary condition, experience indicates that it is a natural and easily satisfied requirement.

Using a new variable z gives us the opportunity to instantiate the accumulator and therefore increase the likelihood that we will be able to match the accumulators in the hypothesis and in the conclusion. Formally, to guarantee that (10) is a useful generalization, we must have

$$\neg p(x) \;\&\; IN(x) \rightarrow IN(n(x))$$

Let R denote the range of a function and D its domain then this condition can be written as

$$[\neg p(y)\&(y \text{ in } D)\&(z \text{ in } D)\&(y' \text{in } R(h1(c,z))] \rightarrow [(g(y,y') \text{ in } R(h1(c,z))]$$

It cannot, of course, be guaranteed that an h1 satisfying (5) will also satisfy the above condition, but the chance that this will happens is always there. In those cases when $h1(c,z) = z$, (5) is always satisfied, namely:

$$[\neg p(y)\&(y \text{ in } D)\&(z \text{ in } D)\&(y' \text{in } D)] \rightarrow [g(y,y') \text{ in } D]$$

This suggests a possible way of finding function h1: depending upon the domain, we know which are identity constants for certain functions (0 is an identity for +, 1 for *, etc), so depending upon the constants certain functions suggest themselves as candidates for h1. The question naturally arises how to find functions h1 and h2 more systematically. The functions h1 and h2 depend on g and very often the function g itself, its some modification or its constituent functions are suitable candidates for h1 and h2. for example if g is such that

$$g(u,g(v,w)) = g(g(u,v),w)$$
$$g(c,z) = z$$

Then h1 is g itself and $h2(u,v,w) = w$. Or, if

$$g(u,g(v,w)) = g(v,g(u,w))$$
$$g(c,z) = z$$

then $h1(u,v) <= g(v,u)$ and $h2(u,v,w) = w$.

At present, we do not know how to find h1 and h2 more systematically, (if they exist at all), but practical experience has convinced us that very often there are natural candidates for such functions. We hope that forthcoming examples will be convincing enough.

Summary

To generalize the accumulator in theorems of the type

$$F1(x,c) = F2(x)$$

where F1 is defined by iteration schema

$$F1(y,y') <= \text{IF } p(y) \text{ THEN } y' \text{ ELSE } F1(f(y),g(y,y')),$$

first we have to find functions h1 and h2 having the properties

$$g(u,h1(v,w)) = h1(g(u,v),h2(u,v,w))$$
$$\text{For some } z\text{: } (h1(u,z) = u \;\&\; h2(u,v,z) = z)$$

Then the generalization is

$$F1(x,h1(c1,z)) = h1(F2(x),F3(x,c1,z))$$

where F3 is defined by

$$F3(y,y',y'') \Leftarrow \text{IF } p(y) \text{ THEN } y'' \text{ ELSE } F3(f(y),g(y,y'),h2(y,y',y''))$$

Remark

Although we have only considered theorems of the type

$$F1(x,c1) = F2(x),$$

all the results and methods can also be used for more complicated cases, e.g.

$$F1(m(x),k(x)) = F2(x)$$

Example 1: Let F1 be defined by

$$F1(y1,y2) \Leftarrow \text{IF } (y1 = 0) \text{ THEN } y2 \text{ ELSE } F1(y1-1,y1*y2)$$

Then

$$g(y1,y2) \Leftarrow y1*y2$$

and we can choose

$$h1(y1,y2) \Leftarrow y1*y2 \qquad h2(y1,y2,y3) \Leftarrow y3.$$

Now (8) is satisfied because

$$g(u,h1(v,w)) = u*(v*w) = (u*v)*w = h1(g(u,v),h2(u,v,w))$$

and condition (11) is satisfied for z=1. The generalization of

$$F1(x,1) = x!$$

is then

$$F1(x,h1(1,z)) = h1(x!,F3(x,1,z))$$

where

```
F3(y1,y2,y3) <= IF (y1=0) THEN y3 ELSE F3(y1-1,y1*y2,y3)
```

Since h2 is the identity function, F3(y1,y2,y3) <= y3. Therefore, substituting for h1, the generalization becomes

```
F1(x,z) = x! * z
```

Example 2: Let F1(y1,y2,y3) be defined by

```
F1(y1,y2,y3) <= IF (y1 = 0) THEN y3
                            ELSE F1(y1 div 2, y2*y2,
                             IF odd(y1) THEN y2*y3 ELSE y3)
```

Then we have

```
f(y1,y2)<= <y1 div 2, y2*y2>
g(y1,y2,y3) <= IF odd(y1) THEN y2*y3 ELSE y3
```

So we can choose h1 and h2 as

```
h1(y1,y2) <= y1*y2          h2(y1,y2,y3,y4) <= y4
```

Now we have

```
g(y1,y2,h1(y3,y4)) = IF odd(y1) THEN y2*(y3*y4)
                                ELSE y3*y4
                   = ( IF odd(y1) THEN y2 *y3 ELSE y3) * y4
                   = h1(g(y1,y2,y3),h2(y1,y2,y3,y4))
```

Therefore

```
F1(x2,x1,1) = x1**x2
```

generalizes to

```
F1(x2,x1,z) = z*(x1**x2).
```

Example 3: Let F1 be defined as follows:

```
F1(y1,y2) <= IF y1 = NIL THEN y2
                         ELSE F1(cdr(y1), car(y1) + 2*y2)
```

Thus g is:

```
g(y1,y2) = car(y1) +2*y2
```

so that

```
g(u,h1(v,w)) = car(u) + 2*h1(v,w).
```

On the other hand, we have

$$h1(g(u,v),h2(u,v,w)) = h1(car(u) + 2*v,h2(u,v,w))$$

So if we choose h1(y1,y2) = y1 + y2 and h2(u,y2,y3) = 2*y3, then we will have

$$g(u,h1(v,w)) = car(u) + 2*(v + w) = car(u) + 2*v +2*w = \\ = h1(g(u,v),h2(u,v,w))$$

F3 is then defined by

```
F3(y1,y2,y3) <= IF y1 = NIL THEN y3
                            ELSE F3(cdr(y1),car(y1)+2*y2,2*y3)
```

or, more simply by

```
F3(y1,y2) <= IF y1 = NIL THEN y2
                         ELSE F3(cdr(y1),2*y2)
```

Thus if the theorem to be proved is

$$F1(x,0) = INTEGER(x)$$

where INTEGER is some standard function translating a list of 0's and 1's into an integer, then this theorem can be generalized into

$$F1(x,w) = INTEGER(x) + F3(x,z)$$

3.2 Generalization Scheme II

Suppose we have to prove

$$F1(x,c,c') = F2(x) \qquad (12)$$

where

```
F1(y1,y2,y3) <= IF (y1=y2) THEN y3
                           ELSE F1(y1,f(y2),g(y2,y3)).
```

To prove this theorem, we have to carry out induction on y2. But this is impossible because the initial value of y is constant. Therefore we must generalize (12) by replacing c with a more general term. In previous heuristics, we replaced a constant with a function h(z) and then tried to find what is the influence of this change of initial value on the final result. It was easy to see how the change of initial value of an accumulator propagates through the whole computation, because of

the limited role played by an accumulator in the function execution. It is harder to apply the same strategy to the recursion variable. The initial value of the accumulator determines the depth of recursion and at each level of recursion, the value of a recursion variable contributes to the final result of the computation. Consequently, the change in the initial value of the induction variable has a much more complicated influence on the final result of the computation. So the choice of the function h must be more careful, and is in fact limited. A good strategy would be to replace c with an expression describing all possible values that the recursion variable can take on during the computation of F1(x,c,c"). Suppose we can derive the values of the recursion variable and the accumulator at the recursion depth z , say h(z) and G(z), respectively. Now if (12) holds, then

$$F1(x,h(z),G(z)) = F2(x)$$

would be a good generalization.

The theorem in (12) could be proven by induction on z (which is, in fact, induction on the depth of recursion). So the question is how to find h(z) and G(z). Observe that

$$F1(x,c,c') = F1(x,f(c),g(c,c')) \quad \text{if } c \neq x$$

$$= F1(x,f^{2}(c),g(f(c),g(c,c'))) \quad \text{if } f(c) \neq x$$

$$\vdots$$

$$= F1(x,f^{i}(c),g(f^{(i-1)}(c),g(f^{(i-2)}(c),\ldots,g(c,c')))) \qquad (13)$$

$$\text{if } i \leq maxi = \min\{j \mid f^{j}(c)=x\}$$

Thus $G(i) = g(f^{(i-1)}(c),g(f^{(i-2)}(c),\ldots,g(c,c')))$

On the other hand if i≤maxi, then

$$F2(f^{i}(x)) = F1(f^{i}(x),c,c')$$

$$= F1(f^{i}(c),f(c),g(c,c'))$$

$$= F1(f^{i}(c), f^{2}(c), g(f(c), g(c,c')))$$

.
.
.

(14)

$$= F1(f^{i}(c), f^{i}(c), g(f^{(i-1)}(c), g(f^{(i-2)}(c), \ldots, g(c,c'))))$$

$$= g(f^{(i-1)}(c), g(f^{(i-2)}(c), \ldots, g(c,c')))$$

$$= G(i)$$

Thus $G(i)$ can be replaced by $F2(f^{i}(c))$

THEOREM 4: Let F1 be defined by (12), and F2 be some previously defined function, then

$$0 \le z \le maxi) \rightarrow F1(x, h(z), F2(h(z))) = F2(x) \qquad (15)$$

holds iff

$$F1(x,c,c') = F2(x)$$

holds, where

$$maxi = \min\{j \mid f^{j}(c) = x\}$$

$$h(z) \Leftarrow \text{IF } z = 0 \text{ THEN } c \text{ ELSE } f(h(z-1))$$

PROOF

<-) by above motivation.
->) On substituting z = 0, (12) is obtained from (15).

The expression (15) is a suitable generalization of (12) if the predicate IN(y1,y2,y3) describing the possible initial values of variables y1, y2 and y3 satisfies the condition

$$\neg p(x) \;\&\; IN(x) \rightarrow IN(n(x)).$$

IN can be defined by

$$IN(y1,y2,y3) \Leftarrow (0 \le z \le maxi) \rightarrow (y2 = h(z) \;\&\; y3 = F2(y2)) \;\&\; (x \text{ and } z \text{ are natural numbers})$$

Intuitively, IN satisfies the condition (5) because it describes

all possible values of variables during the execution of F1(x,c,c'). Formally,

$$IN(y1,y2,y3) \;\&\; (y1 \neq y2) \rightarrow IN(y1,f1(y1),g(y1,y2))$$

If we assume that (12) holds, then the above is really the case which can be proved by substituting the definition of IN and using properties of -> .

Remark :

In the above discussion, we used the simplest case of theorem (12). In fact, the initial value of y2 does not have to be constant. It can also be a function of x, say M(x). But to be able to derive the value of the accumulator at the recursion depth z, it is useful when it is the case that

$$M(h(z,x)) = M(x) \qquad \text{for all } 0 \le z \le maxi$$

where h is defined by

$$h(y1,y2) \Leftarrow IF\ (y2=0)\ THEN\ M(y1)\ ELSE\ f(h(y1,y2-1)).$$

For example, an M(x,z) which has this property is

$$M(y1) \Leftarrow IF\ p(y1)\ THEN\ y1\ ELSE\ M(If(y1)),$$

where If is inverse of f, i.e. f(If(y1)) = y1. Now we can write

$$M(x) = M(If(x)) = M(If^{2}(x)) = \ldots = M(If^{maxi}(x)) = If^{maxi}(x)$$

$$h(x,z) = f^{z}(If^{maxi}(x)) = If^{(maxi-z)}(x), \qquad \text{for all } 0 \le z \le maxi$$

$$\text{and } M(h(x,z)) = M(If1^{(maxi-z)}(x)) = If1^{(maxi)}(x) = M(x,z)$$

We can therefore generalize F(x,M(x),c') = F2(x) into

$$0 \le z \le maxi \rightarrow F1(x,h(x,z),F2(h(x,z)) = F2(x)$$

and, using the relation between f and If, this can be rewritten

$$(0 \le z \le maxi') \rightarrow F1(x,h'(x,z),F2(h'(x,z)) = F2(x),$$

where

$$h'(x,z) \Leftarrow IF\ z = 0\ THEN\ x\ ELSE\ If1(h'(x))$$
$$maxi' = \min\{i \mid p(h'(x,i))\}.$$

Example 4

```
F1(y1,y2,y3) <= IF (y1 = y2) THEN y3
                             ELSE F1(y1,y2+1,(y2+1)*y3)
```

We define h as follows

```
h(z) <= IF z = 0 THEN 0 ELSE h(z-1) + 1
```

or in other words, h(z) <= z. So we can generalize F1(x,0,1) = x! into

```
(0≤z≤x) -> F1(x,z,z!) = x!
```

because maxi = min{i | (i = x) } = x.

Example 5: Let div denote integer division and mod denote the remainder after the integer devision.

```
M(y1,y2) <= IF y1 < y2 THEN y2 ELSE M(y1,2*y2)
F1(y1,y2,y3,y4) <= IF (y1 = y2 THEN <y3,y4>
                   ELSE IF(y3 ≥ y2 div 2)
                      THEN F1(y1,y2div2,y3-y2div2,2*y4+1)
                      ELSE F1(y1,y2div2,y3,2*y4)
```

Since (2*y2) div 2 = y2, we can apply the above method, generalizing

```
F1(x2,M(x1,x2),x1,0) = <x1 mod x2, x1 div x2>
```

into

```
          (0≤z≤maxi) ->
F1(x2,h(x2,z),x1modh(x2,z),x1divh(x2,z)) = <x1modx2,x1divx2>
```

where h(y1,y2) <= IF y2 = 0 THEN y1 ELSE 2*h(y1,y2-1).

In a more standard notation, $h(y1,y2) = 2^{y2} * y1$,

and maxi is then $\min\{i \mid 2^{i} * x2 > x1\}$

4. REDEFINITIONS

The reason why some theorems are not strong enough to carry themselves through induction is that the input specification is not general enough; that is the condition

$$\neg p(x) \;\&\; IN(x) \rightarrow IN(n(x))$$

is not satisfied. Theorem generalization removes this limitation by modifying IN. Another possible strategy is to try to redefine the function. We now describe how redefinition can be performed for two classes of programs.

Consider the example F(x,1) = x! ,where F is defined by

F1(y1,y2) <= IF y1=0 THEN y2 ELSE F1(y1-1,y1*y2))

The function F has two arguments, but we are interested in the behavior of F with one argument (y2) very specialized in its use: its initial value is constant. For all purposes, F has degenerated into a one-variable function. So if we can translate F(x1,1) into a true one-argument function, it would become easier to manage.

Let F(x1,1) = F'(x1) where

F'(y1) <= IF y1=0 THEN 1 ELSE y1*F'(y1-1)

and let the theorem to prove be

$$F'(x1) = x1!$$

Now if F'(x) = x!, then we have

F'(x1+1)	
= (x1+1)*F'(x1)	from the def. of F'
= (x1+1)*x1!	from the hypothesis
= (x1+1)!	property of !

4.1 Redefinition Scheme I

THEOREM 5: Let F1 be defined by iteration.

F1(y,y') <= IF p(y) THEN y' ELSE F(f(y),g(y,y'))

Suppose we can find the functions h1 and h2 with the properties

$$g(z,c) = h1(c,z) \qquad (16)$$

and

$$g(u,h1(v,w)) = h1(g(u,v),h2(w))$$

Then we can define functions F and F3 by

```
F(y) <= IF p(y) THEN c
                ELSE h1(F(f(y)), F3(f(y),y))
F3(y,y') <= IF p(y) THEN y'
                    ELSE F3(f(y),h2(y'))
```

such that

$$F1(x,c) = F(x)$$

PROOF: By case induction.

Basis: Assume p(x) is true, then from the definitions of F and F3 it follows that

$$F(x) = c = F1(x,c).$$

Induction step:
Assume ¬p(x) and F(f(x)) = F1(f(x),c) then

```
  F(x) = h1(F(f(x)),F3(f(x),x)      from def. of F
= h1(F1(f(x),c),F3(f(x),c))         from assumption
  F1(x,c) = F1(f(x),g(x,c))         from def. of F1
= F1(f(x),h1(c,x))              from property (16)
```

Thus we have to prove

$$F1(f(x),h1(c,x)) = h1(F1(f(x),c),F3(f(x),c)).$$

But this is just an instance of LEMMA 1 with x instantiated as f(x), w as c, and z as x.

Example 6

```
F1(y1,y2) <= IF y1 = NIL THEN y2
                         ELSE F(cdr(y1),APPEND(y2,car(y1))))
APPEND(y1,y2)<=IF atom(y1) THEN y2
                        ELSE cons(car(y1),APPEND(cdr(y1),y2)))
```

and we have to redefine

$$F1(x,NIL)$$

We proceed by defining

g(y1,y2) <= APPEND(y2,car(y1))

We can choose h1 and h2 as

h1(y1,y2) <= APPEND(y2,y1) and h2(y1,y2,y3) <= y3

because

g(u,h1(v,w)) = APPEND(APPEND(w,v),car(u))
= APPEND(w,APPEND(v,car(u))) = h1(g(u,v),h2(u,v,w)).

Also

g(x,nil) = APPEND(x,NIL) = h1(NIL,x)

Thus F1(x,NIL) can be redefined as

F(y) <= IF y = NIL THEN NIL ELSE APPEND(car(y),F(cdr(y)))

4.2 Redefinition Scheme II

THEOREM 6: Let F1 be defined by

F1(y,y') <= IF p(y) THEN G(y,y') ELSE F1(f(y),y')

where

G(y,y') <= IF q(y) THEN y' ELSE G(If(y),g(y,y')).

Provided that we have

$$If(f(x)) = x \tag{17}$$

$$\max\{j \mid q(f^{j}(x))\} = 0 \tag{18}$$

we can transform F1(x,c) into F(x), where

F(y) <= IF p(y) THEN c ELSE g(f(y),F(f(y))).

PROOF

Let $maxi = \min\{j \mid p(f^{j}(x))\}$.

From the definition it follows that

$$Fl(x,c) = Fl(f(x),c) = Fl(f^{2}(x),c) = \ldots = Fl(f^{maxi}(x),c)$$

$$= G(f^{maxi}(x),c)$$

Because of (18) it also follows that

$$G(f^{maxi}(x),c) = G(\,If(f^{maxi}(x)),g(f^{maxi}(x),c))$$

$$= G(f^{(maxi-1)}(x),g(f^{maxi}(x),c))$$

$$\vdots$$

$$= G(x,g(f(x),g(f^{2}(x),\ldots,g(f^{maxi}(x),c))))$$

$$= g(f(x),g(f^{2}(x),\ldots,g(f^{maxi}(x),c)))$$

Now we can evaluate F(x) as follows:

$$F(x) = g(f(x),F(f^{2}(x))) = g(f(x),g(f(x),F(f(x))))$$

$$= \ldots = g(f(x),g(f^{2}(x),\ldots,g(f^{maxi}(x),F(f^{maxi}(x)))))$$

$$= g(f(x),g(f^{2}(x),\ldots,g(f^{maxi}(x),c)))$$

Hence we conclude that Fl(x,c) = F(x).

Suppose that instead of knowing the complete definition of Fl we have only a definition of G. We can rewrite G provided we can find suitable f and p. The function f must satisfy the properties (17) and (18), and, moreover, p must also satisfy the condition

$$\min\{j \mid p(f^{j}(x)\} = \min\{j \mid q(If^{j}(x))\} \qquad (19)$$

In some cases it is possible to find such p and f quite easily. For example let G be defined as

```
G(y1,y2,y3) <= IF y1 = y2 THEN y3 ELSE G(y1,If(y2),g(y2,y3))
```

and let us try to redefine G(x,c,c"). Suppose we can find f such that

$$If(f(x)) = f(If(x)) = x$$

then

$$p(y) <= (y = c).$$

We can prove by contradiction that p(y) satisfies (19).

Let $k = \min\{j \mid If^{j}(c)) = x\}$

Suppose there is a $m < k$ such that $c = f^{m}(x)$, then

$$If(c) = f^{(m-1)}(x)$$

$$If^{2}(c) = f^{(m-2)}(x)$$

.
.
.

$$If^{m}(c) = x$$

which is contrary to our assumption.

Example 7

```
F1(y0,y1,y2,y3) <= IF y0 < y2 THEN G(y0,y1,y2,y3)
                               ELSE F1(y0,y1,2*y2,y3)

G(y0,y1,y2,y3) <= IF y1 = y2 THEN y3
                          ELSE IF (y3≥y2div2)
                               THEN G(y0,y1,y2div2,y3-y2div2)
                               ELSE G(y0,y1,y2div2,y3).
```

We will redefine F1(x1,x2,x2,x1). Since

$$y = \langle y0,y1,y2\rangle,$$
$$f(y) = \langle y0,y1,2*y2\rangle,$$
$$If(y) = \langle y0,y1,y2\underline{div}2\rangle,$$

the conditions (17) and (18) are satisfied:

$$If(f(y)) = If(\langle y0, y1, y2*2\rangle = \langle y0, y1, (y2*2)\underline{div}2\rangle = y$$

$$\max\{j \mid x2 = 2^{j} * x2\} = 0$$

The function F is then

```
F(y0,y1,y2) <= IF y0 < y2 THEN x1
                ELSE IF (F(y0,y1,2*y2)≥(2*y2)div2)
                     THEN F(y0,y1,2*y2)-(2*y2)div2
                     ELSE F(y0,y1,2*y2).
```

We can simplify this definition to

```
F(y1,y2) <= IF y1 < y2 THEN y0
            ELSE IF (F(y1,2*y2)≥y2 )
                 THEN F(y1,2*y2) - y2
                 ELSE F(y1,2*y2).
```

Example 8

```
F1(y1,y2,y3) <= IF y1=y2 THEN y3 ELSE F1(y1,y2+1,(y2+1)*y3)
```

We would like to redefine F1(x,0,1). Since If(y2) <= y2+1, we can define f(y2) <= y2-1. Now we have

```
If(f(y2)) = If(y2+1) = (y2+1)-1 = y2 = (y2-1)+1 = f(If(y2))
```

Therefore p(y2), which satisfies the requirement (19), is

```
p(y2) <= (y2 = 0)
```

and the new definition of F1(x,0,1) is

```
F(y1) <= IF y1 = 0 THEN 1 ELSE ((y-1)+1)*F(y-1).
```

Or, after some simplifications, we can define

```
F(y1) <= IF y1 = 0 THEN 1 ELSE y*F(y1-1).
```

5. CONCLUSION

We have developed some methods for generalizing theorems about recursively defined functions, so that the generalized form of these theorems is more suitable for proof by induction. We have given some heuristics to carry out the generalization for certain patterns of theorems and recursive definitions. Invariant assertions are sometimes obtained as a by-product in

this generalization.

Our generalization are heuristics are based on an analysis of defined functions. Whenever these heuristics are applicable, the generalized theorems are true iff the original theorems are. This is not the case with the heuristics in Boyer and Moore [2]. In their heuristics (replacing the terms common to both sides of equality or implication), the above relation is missing. Furthermore, our heuristics are based on an analysis of definitional schemas. Given a function, we find a matching schema and obtain information about the function that will enable us to generalize. Aubin [1], on the other hand, analyses functions ad hoc to replace a constant with an expression describing all the possible values this argument could acquire. Our choice of expressions to replace constants, however, takes into account the influence this change of the initial value will have on the final result of the computation.

We believe it is possible to apply our generalization method to other types of definitional schemas and develop a catalog of heuristics for different classes of programs. This seems more useful than generalizing the same heuristics for very large class of programs, since that would complicate the test of the heuristic's applicability.

In the literature, the work on redefinition of functions has been done for other purposes: e.g. when defining functions by recursion, one may try to find a more compact definition of composition of such functions (Chatelin [3]). We have given methods to redefine functions in order, again, to simplify the proof of certain theorems describing the properties of recursively defined functions. With these redefined functions, theorems become much easier to prove than with the original definitions.

REFERENCES

1. Aubin, R. 1975: "Some Generalization Heuristics in Proofs by Induction.", Colloques IRIA Proving and Improving Programs, Arc et Senans, pp.197-208 (July)

2. Boyer, R.S. and Moore,J.S. 1975: "Proving Theorems About LISP Functions", Journal of A.C.M. 22, 1, pp.129-174 (January)

3. Chatelin,P. 1977: "Self-redefinitions as a Program Manipulation Strategy", SIGPLAN Notices 12,8, pp.174-179 (August)

4. Greif,I. and Waldinger, R.J. 1974: "A More Heuristic Approach to Program Verification", Proc. Intl. Symp. on Programming, Paris, France, pp.83-90

5. Manna,Z. 1974: "Mathematical Theory of Computation" McGraw-Hill Book Co.,New York, NY.

6. Manna,Z. and Pnuelli,A. (1970) : "Towards a Mathematical Theory of Computation",Journal of ACM 17, 3, pp.555-569 (July)

7. McCarthy,J.1962: " Towards a Mathematical Science of Computation", Information Processing, Proceedings of IFIP Congress 1962, (ed., C.M. Popplewell), North Holland Publishing Company, Amsterdam, pp.21-28

8. Misra,J. 1975: "Relations Uniformly Conserved by a Loop", Colloques IRIA Proving and Improving Programs, Arc et Senans, pp.71-79, (July)

9. Moore,J.S. 1974: "Introducing Iteration into the Pure LISP Theorem Prover", CSL-74-3, Xerox Palo Alto Research Center, Palo Alto,Ca.

10. Morris,H.J . and Wegbreit,B. 1977: "Subgoal Induction", Communications of ACM 20,4, pp.208-222 (April) Programs", Ph.D. Thesis, University of Edinburgh, Edinburgh

11. Wegbreit,B. 1974: " The Synthetis of Loop Predicates", Comm. of the ACM 17, 2, pp.102-112, (February)

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CS-8102	2. GOVT ACCESSION NO. AD-A103749	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THEOREM GENERALIZATION IN PROGRAM VERIFICATION		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jan Vytopil S. Kamal Abdali		8. CONTRACT OR GRANT NUMBER(s) ONR N00014-75-C-1026
9. PERFORMING ORGANIZATION NAME AND ADDRESS Mathematical Sciences Department Rensselaer Polytechnic Institute Troy, N. Y. 12181		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Resident Representative 715 Broadway-5th Floor, N.Y., N.Y. 10003		12. REPORT DATE June 1981 13. NUMBER OF PAGES 31
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Program verification, theorem generalization, invariant assertion

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

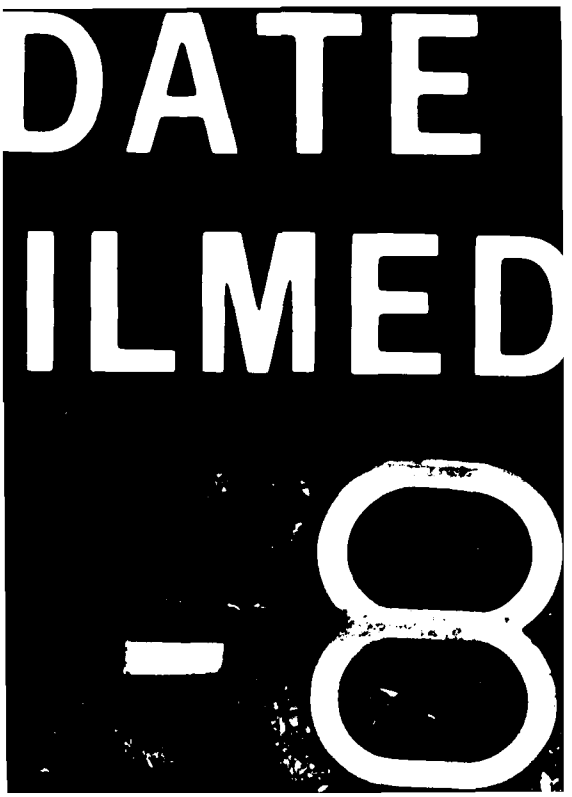
(see back-side of page)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The generalization of theorems about programs obtained from recursive schemas is discussed. Methods are given to generalize theorems about two classes of programs to make the theorems easier to prove by induction. Invariant assertions are obtained as a by-product of the generalization process. Also, methods are given to redefine the functions representing programs so as to simplify the proof of programs properties in certain cases.


DATE
ILMED
-8